Syntax Errors Investigation System for PHP Program using Top-Down Parsing Approach

Su Su Win
Faculty of Computer Science
University of Computer Studies (Mandalay)
susuwin.loikaw@gmail.com

Zar Zar Tun
Faculty of Information Science
University of Computer Studies (Taunggyi)
zarzarhtun31@gmail.com

Abstract

Parsing is a necessary mechanism for many natural language processing applications, such as machine translation, question answering, knowledge extraction and information retrieval. In this paper, we propose the syntax investigation system which can investigate the syntax errors in the PHP program. This system takes each string from a PHP program as input and determines it as a correct message or error message by using the top down parsing approach. The predictive parser is used to construct the parse table, which determines the action of the input string based on the grammar production rules. These grammar rules were reconstructed if the left recursion rules are occurred to avoid the left recursion problem in the top down parsing approach. As a result, our system can reproduce the correct messages, error messages, and location errors. It was confirmed that our system can be useful to investigate the syntax effectively in PHP programming language.

Keywords: Top-Down Parsing, Predictive Parser, Left recursion.

1. Introduction

Parsing in basic terms can be described as breaking down the sentence into its constituent words in order to find out the grammatical type of each word or alternatively to decompose an input into more easily processed components. In simple terms parsing is breaking down of sentence into atomic values. Parsing is a process of determining how a string of terminals (sentence) is generated from its constituents, by breaking down of sentence into tokens. Each individual word in a sentence is termed as token. A token of a language is a category of its lexemes. For example, an identifier is a token that can have lexemes, or instances, such as id_name.

This paper is organized as followings: We analyze the previous efforts related to the tasks of parsing in section 2. Section 3 explains the procedure of proposed system. Section 4 describes the top down parsing approach and grammar rules. Section 5 explains about experimental results. Finally, the conclusion of the paper is presented.

2. Related Work

Ahmad AI-Taani, Mohammed Msallam and Sana Wedian [1] presented an efficient top-down chart parser for parsing simple Arabic sentences.

Win Win Thant, Tin Myat Htwe and Ni Lar Thein [2] described a context-free grammars (CFG) based top-down parsing for Myanmar sentences.

K.M.Azharul Hasan, AI-Mahmud, Amit Mondaal and Amit Saha [3] presented Context-free grammars (CFG) for Bangla language and proposed a Bangla parser based on the grammar. Rachana Rangra and Madhusudan [4] described the parsing techniques in natural language processing.

Win Win Thant, Tin Myat Htwe and Ni Lar Thein [5] described the use of Naïve Bayes to address the task of assigning function tags and context-free grammars (CFG) to parse Myanmar sentences.

3. Proposed System

Our proposed system is shown in Figure 1. The system accepts each string from a program and determines it as a correct message or error by using the top down parsing approach. The accepted input stream is parsed based on the predictive parsing program and the parse table is created. The parse table checks the input string based on the grammar production rules, which were already reconstructed by eliminating the left recursive rules. If the input string is matched with grammar rules, the correct message is displayed as a parse tree. If the input string is not matched with the grammar rules, the error messages and location errors are displayed.

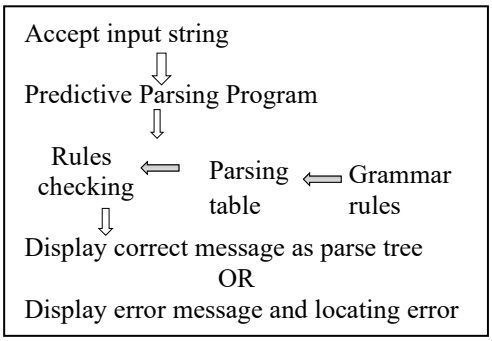

Figure 1. Our proposed system

4. Top Down Parsing Approach

We use the top-down parsing approach, which builds the parse tree from top to down order. The parser reads the sequence of input tokens from left to right, and it builds the parse tree according to the same order. The parser builds the parse tree from the top node of the tree as the start nonterminal symbol and expands the leftmost nonterminal symbol until the child node is terminal symbol.

4.1. Producing Grammar Rules

Firstly, we produce the grammar rules, which are used to check the input string to confirm the error or correct message. To produce the grammar rules, a context-free grammar (CFG) is used to specify the syntax of the programming language. The CFG consists of rules (production), terminal (token), non-terminal, and a start symbol. The grammar rules are produced from a PHP sample program as shown in Table 1.

Table 1. Grammar rules for a PHP sample program

<program> ⟶ <open_tag><block_stmt> <close_tag>
<open_tag> ⟶ <open_tag> <?PHP |<?PHP |<open_tag> <? | <?
<close_tag> ⟶ <close_tag>?> | ?>
<block_stmt> ⟶<dec_list> <stmt_list>| <stmt_list>
<dec_list> ⟶ <$id_list> ;
<$id_list> ⟶ <$id_list> , $ID | $ID
<stmt_list> ⟶<stmt_list> <stmt> | <stmt>
<stmt> ⟶ <assign> | <write> | <if> | <for> | <while> | <do_while>|<expression>
<assign> ⟶ $ID = <expr> ;
<expr> ⟶<expr> + <term> |<expr> - <term> | <expr> /<term> | <term>
<term> ⟶ <term> * <factor> | <factor>
<factor> ⟶ $ID | $NUM
<write> ⟶ ECHO <write_list> ;
<write_list>⟶ $LITERAL | $ID
<if> ⟶ IF (<expression>) <stmt>
<expression> ⟶<expr> <logical> <expr>
<logical> ⟶ < | > | <= | >= | != | ==|+=|-=|*=|/=
<for> ⟶ FOR (<assign> ; <expression> ; <counter>) { <stmt_list> }
<while> ⟶ WHILE (<expression>) { <stmt_list> }
<do_while> ⟶ DO { <stmt_list> } WHILE (<expression>);
<operator> ⟶ ++ | --
<counter> ⟶$ID <operator>

Some of the grammar rules in bold face listed in table 1 include the left recursive grammar, which often poses infinite recursion problem. Therefore, the grammar rules are reconstructed by eliminating the left recursion rules using the formal technique.

We can transform the grammar as follows:

Rule : Replace with

A-> β1A′| β2A′|…..| βnA′

A-> α1A′| α2A′|…..| βmA′|ε

The grammars rules in table 1 are repaired by eliminating the left recursion rules, and the grammar production rules are reconstructed as shown in Table 2.

Table 2. Grammar rules after eliminating the left recursive rules

<program>⟶<open_tag><block_stmt><close_tag>
<open_tag>⟶<?PHP<open_tag'>| <? <open_tag'>
<open_tag'> ⟶ <?PHP <open_tag'> | Ɛ
<open_tag'> ⟶ <? <open_tag'> | Ɛ
<close_tag> ⟶ ?> <close_tag'>
<close_tag'> ⟶?> <close_tag'>| Ɛ
<block_stmt>⟶ <dec_list> <stmt_list>|<stmt_list>
<dec_list> ⟶<$id_list> ;
<$id_list> ⟶$ID <$id_list'>
<$id_list'> ⟶, $ID <$id_list'> |Ɛ
<stmt_list> ⟶<stmt> <stmt_list'>
<stmt_list'> ⟶<stmt> <stmt_list'> |Ɛ
<stmt> ⟶<assign> | <write> | <if> | <for> | <while> | <do_while>|<expression>
<assign> ⟶ $ID = <expr> ;
<expr> ⟶ <term> <expr'>
<expr'> ⟶ + <term> <expr'> |Ɛ
<expr'> ⟶ -<term> <expr'> |Ɛ
<expr'> ⟶ / <term> <expr'> |Ɛ
<term> ⟶ <factor> <term'>
<term'> ⟶ * <factor> <term'> | Ɛ
<factor> ⟶ $ID | $NUM
<write> ⟶ ECHO<write_list>;|ECHO"<write_list>";
write_list> $LITERAL | $ID
<if> ⟶ IF (<expression>) <stmt>|
IF (<expression>) <stmt> ELSE <stmt>
<expression>⟶ <expr> <logical> <expr>
<logical> ⟶< | > | <= | >= | != | ==|+=|-=|*=|/=
<for> ⟶ FOR (<assign> ; <expression> ; <counter>) { <stmt_list> }
<while> ⟶ WHILE (<expression>) { <stmt_list> }
<do_while> ⟶ DO { <stmt_list> } WHILE (<expression>);

In table 2, some non-terminal symbols in bold face have two or more distinct definitions representing two or more possible syntactic forms. Therefore, these rules are broken down into individual rules in table.

Table 3. Produced possible grammar rules

R1.<program> ⟶ <open_tag><block_stmt> <close_tag>
R2.<open_tag> ⟶ <?PHP <open_tag'>
R3. <open_tag> ⟶ <? <open_tag'>
R4. <open_tag'> ⟶ <?PHP <open_tag'>
R5. <open_tag'> ⟶ <? <open_tag'>
R6.<open_tag'> ⟶ Ɛ
R7.<close_tag> ⟶ ?> <close_tag'>
R8.<close_tag'> ⟶ ?> <close_tag'>
R9.<close_tag'> ⟶ Ɛ
R10. <block_stmt> ⟶ <dec_list> <stmt_list>
R11. <block_stmt> ⟶ <stmt_list>
R12.<dec_list> ⟶ <$id_list> ;
R13. <$id_list> ⟶ $ID <$id_list'>
R14. <$id_list'> ⟶ , $ID <$id_list'>
R15.<$id_list'> ⟶ Ɛ
R16. <stmt_list> ⟶ <stmt> <stmt_list'>
R17. <stmt_list'> ⟶ <stmt> <stmt_list'>
R18. <stmt_list'> ⟶ Ɛ
R19 .<stmt> ⟶ <assign>
R20. <stmt> ⟶ <write>
R21. <stmt> ⟶ <if>
R22. <stmt> ⟶ <for>
R23. <stmt> ⟶ <while>
R24. <stmt> ⟶ <do_while>
R25. <stmt> ⟶ <expression>
R26.<assign> ⟶ $ID = <expr> ;
R27.<expr> ⟶ <term> <expr'>
R28.<expr'> ⟶ + <term> <expr'>
R29.<expr'> ⟶ -<term> <expr'>
R30.<expr'> ⟶ / <term> <expr'>
R31. <expr'> ⟶ Ɛ
R32. <term> ⟶ <factor> <term'>
R33. <term'> ⟶ * <factor> <term'>
R34. <term'> ⟶ Ɛ
R35. <factor> ⟶ $ID
R36. <factor> ⟶ $NUM
R37. <write> ⟶ ECHO <write_list> ;
R38. <write> ⟶ ECHO "<write_list> ";
R39. <write_list> ⟶ $LITERAL
R40. <write_list> ⟶ $ID
R41.<if> ⟶ IF (<expression>) <stmt>
R42.<if> ⟶ IF (<expression>) <stmt>ELSE<stmt>
R43.<expression> ⟶ <expr> <logical> <expr>
R44. <logical> ⟶ <
R45. <logical> ⟶ >
R46. <logical> ⟶ <=
R47. <logical> ⟶ >=
R48. <logical> ⟶ !=
R49. <logical> ⟶ ==
R50. <logical> ⟶ +=
R51. <logical> ⟶ -=
R52. <logical> ⟶ *=
R53. <logical> ⟶ /=
R54. <for> ⟶ FOR (<assign> ; <expression> ; <counter>) { <stmt_list> }
R55 .<while> ⟶ WHILE (<expression>){ <stmt_list> }
R56.<do_while> ⟶ DO{ <stmt_list>}WHILE (<expression>);
R57 .<operator> ⟶ ++
R58 .<operator> ⟶ --
R59. <counter> ⟶ $ID <operator>

In table 3, all possible grammar rules are produced, and it is easier to create the parse table.

4.2. Predictive Parser

A parser takes an input string in the form of the sequence of tokens and produces the output in the form of parse tree or an error message. We use the predictive parser, which uses a stack and a parsing table to parse the input string and generate a parse tree. To construct a predictive parser, two functions namely FIRST () and FOLLOW () are important. The rules for first sets are as follows:

1. If **a** is a terminal, then FIRST (A) = {'a'}

2. If A->ϵ is a production rule, then add ϵ to FIRST (A).

3. If A->B1 B2 B3…Xn is a production,

1. FIRST (A) =FIRST (B1)

2. If FIRST (B1) contains then FIRST (A) = {FIRST (B1) - ϵ }U{FIRST (B2)}

3. If FIRST (Bi) contains for all i=1 to n , then add ϵ to FIRST (A).

The grammar rules are constructed according to the first set is shown in table 4.

Table 4. Grammar Rules by First Sets

FIRST (program) = {<?PHP,<?}
FIRST (open_tag)={<?PHP,<?}
FIRST(open_tag')={<?PHP,<?, Ɛ}
FIRST (close_tag)={?>}
FIRST (close_tag')={?>,Ɛ}
FIRST (block_stmt)={$ID,ECHO,IF,FOR,WHLIE,DO}
FIRST (dec_list)={$ID}
FIRST($id_list)={$ID}
FIRST($id_list')={,Ɛ}
FIRST (stmt_list)={$ID,ECHO,IF,FOR,WHLIE,DO}
FIRST (stmt_list')={$ID,ECHO,IF,FOR,WHLIE,DO,Ɛ}
FIRST (stmt)={$ID,ECHO,IF,FOR,WHLIE,DO}
FIRST (assign)={$ID}
FIRST (expr)={$ID,$NUM}
FIRST (expr')={+,-,/,Ɛ}
FIRST (term)={$ID,$NUM}
FIRST (term')={*,Ɛ}
FIRST (factor)={$ID,$NUM}
FIRST (write)={ECHO}
FIRST (write_list)={$LITERAL,$ID}
FIRS T(if)={IF}
FIRST (expression)={$ID,$NUM}
FIRST (logical)={<,>,<=,>=,!=,==,+=,-=,*=,/=}
FIRST (for)={FOR}
FIRST (while)={WHLIE}
FIRST (do_while)={DO}
FIRST (operator)={++,--}
FIRST (counter)={$ID}

The rules for follow sets are as follows:

1. FOLLOW(S) = {$} // where S is the starting Non-Terminal

2. If X -> pYq is a production, where p, Y and q are any grammar symbols, then everything in FIRST (q) except Є is in FOLLOW (Y).

3. If X->pY is a production, then everything in FOLLOW (X) is in FOLLOW (Y).

4. If X->pYq is a production and FIRST (q) contains Є, then FOLLOW (Y) contains {FIRST (q) – Є} U FOLLOW (X)

The grammar rules are constructed according to the first set and then, the predictive parsing table is created by using the grammar rules in table 3, grammar rules by first sets in table 4, and grammar rules by follow sets in table 5 to check the input token streams.

Table 5. Predictive Parsing Table

Non terminal	Input Token (Terminal)	Rules	Error Message For Non terminal and Input Tokens
<program>	<?PHP	R1	Excepted <?PHP , <?
<program>	<?	R1	
<open_tag>	<?PHP	R2	Excepted <?PHP, <?
<open_tag>	<?	R3	
<open_tag'>	<?PHP	R4	Excepted <?PHP , <?
<open_tag'>	<?	R5	
<open_tag'>	ε	R6	Skip
<close_tag>	?>	R7	Excepted ?>
<open_tag'>	?>	R8	Excepted ?>
<open_tag'>	ε	R9	Skip
<block_stmt>	$ID	R10	Excepted $ID,$ID,ECHO,$IF, FOR, WHILE,DO
<block_stmt>	$ID	R11	
<block_stmt>	ECHO	R11	
<block_stmt>	IF	R11	
<block_stmt>	FOR	R11	
<block_stmt>	WHLIE	R11	
<block_stmt>	DO	R11	
<dec_list>	$ID	R12	Excepted $ID
<$id_list>	$ID	R13	Excepted $ID
<$id_list'>	,	R14	Excepted ,
<$id_list'>	ε	R15	Skip
<stmt_list>	$ID	R16	Excepted $ID,ECHO,IF, FOR,WHLIE,DO
<stmt_list>	ECHO	R16	
<stmt_list>	IF	R16	
<stmt_list>	FOR	R16	
<stmt_list>	WHLIE	R16	
<stmt_list>	DO	R16	
<stmt_list'>	$ID	R17	Excepted $ID,ECHO,IF,
<stmt_list'>	ECHO	R17	
<stmt_list'>	IF	R17	
<stmt_list'>	FOR	R17	Excepted FOR,WHLIE,DO
<stmt_list'>	WHLIE	R17	
<stmt_list'>	DO	R17	
<stmt_list'>	ε	R18	Skip
<stmt>	$ID	R19	Excepted $ID,ECHO,IF, FOR,WHILE,DO,$ID,$NUM
<stmt>	ECHO	R20	
<stmt>	IF	R21	
<stmt>	FOR	R22	
<stmt>	WHLIE	R23	
<stmt>	DO	R24	
<stmt>	$ID	R25	
<stmt>	$NUM	R25	
<assign>	$ID	R26	Excepted $ID
<expr>	$ID	R27	Excepted $ID,$NUM
<expr>	$NUM	R27	
<expr'>	+	R28	Excepted +,-, /
<expr'>	-	R29	
<expr'>	/	R30	
<expr'>	ε	R31	Skip
<term>	$ID	R32	Excepted $ID,$NUM
<term>	$NUM	R32	
<term'>	*	R33	Excepted *
<term'>	ε	R34	Skip
<factor>	$ID	R35	Excepted $ID,$NUM
<factor>	$NUM	R36	
<write>	ECHO	R38	
<write_list>	$LITERAL	R39	
<write_list>	$ID	R40	Excepted $LITERAL,$ID
<if>	IF	R41	Excepted IF,IF
<if>	IF	R42	
<expression>	$ID	R43	Excepted $ID,$NUM
<expression>	$NUM	R43	
<logical>	<	R44	Excepted <,>,<=,>= ,!=,==,+=, -=,*=,/=
<logical>	>	R45	
<logical>	<=	R46	
<logical>	>=	R47	
<logical>	!=	R48	
<logical>	==	R49	
<logical>	+=	R50	
<logical>	-=	R51	
<logical>	*=	R52	
<logical>	/=	R53	
<for>	FOR	R54	Excepted FOR
<while>	WHLIE	R55	Excepted WHLIE
<do_while>	DO	R56	Excepted DO
<operator>	++	R57	Excepted ++,--
<operator>	--	R58	
<counter>	$ID	R59	Excepted $ID

The predictive parser uses the parse table, grammar rules in table 3, and the flowing parsing algorithm to check the state of the input string.

parsing algorithm:

set **a** be the first symbol of w;
set A to the top stack symbol;
while (A≠$) { /* stack is not empty */
if (A = a) pop the stack and
let a be the next symbol of w ;
if (A is a terminal) error();
if (M [A, a] is an error entry) error();
if (M [A, a] = A → B1 B2 … Bk) {
output the production A → B1 B2 … Bk;
pop the stack;
push Bk, Bk-1, … B1 onto the stack, with B1 on top;
}
if A=$,Sentence is Accepted.
}

The Table 7 shows the moves made by top down parser for the following PHP sample input token streams.
<? PHP $ID;$ID=$NUM;ECHO $ID; ?>

The pointer reads the input string character by character. If the stack and at the end of the input string contain an end symbol **$**, it is denoted that the stack is empty, and the input is also consumed.

Table 6. Moves made by Top-Down Parser

Stack	Input	Action
$<program>	<?PHP $ID; $ID=$NUM; ECHO $ID; ?>$	R1
$<close_tag><block_stmt> <open_tag>	<?PHP $ID;$ID=$NUM;E CHO $ID; ?>$	R2
$<close_tag><block_stmt> <open_tag'><?PHP	<?PHP $ID;$ID=$NUM;E CHO $ID; ?>$	Terminal
$<close_tag><block_stmt> <open_tag'>	$ID;$ID=$NUM;E CHO $ID;?>$	R6 Ɛ
$<close_tag><block_stmt>	$ID;$ID=$NUM;E CHO $ID;?>$	R10
$<close_tag><stmt_list> <dec_list>	$ID;$ID=$NUM $ID;$ID=$NUM;E CHO $ID;?>$	R12
$<close_tag><stmt_list> ; <$id_list>	$ID;$ID=$NUM;E CHO $ID;?>$	R13
$<close_tag><stmt_list>; <$id_list'> $ID	$ID;$ID=$NUM;E CHO $ID;?>$	Terminal
$<close_tag><stmt_list>; <$id_list '>	;$ID=$NUM; ECHO $ID;?>$	R15 Ɛ
$<close_tag><stmt_list>;	;$ID=$NUM; ECHO $ID;?>$	Terminal
$<close_tag><stmt_list>	$ID=$NUM;ECHO $ID;?>$	R16
$<close_tag><stmt_list'> <stmt>	$ID=$NUM;ECHO $ID;?>$	R19
$<close_tag><stmt_list'> <assign>	$ID=$NUM;ECHO $ID;?>$	R26
$<close_tag><stmt_list'>; <expr>=$ID	$ID=$NUM; ECHO $ID;?>$	Terminal
$<close_tag><stmt_list'>; <expr>=	=$NUM;ECHO $ID;?>$	Terminal
$<close_tag><stmt_list'>; <expr>	$NUM; ECHO $ID ;?> $	R27
$<close_tag><stmt_list'>; <expr'><term>	$NUM;ECHO $ID;?>$	R32
$<close_tag><stmt_list'>; <expr'><term'><factor>	$NUM;ECHO $ID;?>$	R36
$<close_tag><stmt_list'>; <expr'><term'>$NUM	$NUM;ECHO $ID;?>$	Terminal
$<close_tag><stmt_list'>; <expr'><term'>	; ECHO $ID;?>$	R34 Ɛ
$<close_tag><stmt_list'>; <expr'>	; ECHO $ID;?>$	R31 Ɛ
$<close_tag><stmt_list'>;	;ECHO $ID;?>$	Terminal
$<close_tag><stmt_list'>	ECHO $ID;?>$	R17
$<close_tag><stmt_list'> <stmt>	ECHO $ID;?>$	R20
$<close_tag><stmt_list'> <write>	ECHO $ID;?>$	R37
$<close_tag><stmt_list'> ECHO	ECHO $ID;?>$	Terminal
$<close_tag><stmt_list'>; <write_list>	$ID;?>$	R40
$<close_tag><stmt_list'>; $ID	$ID;?>$	Terminal
$<close_tag><stmt_list'>;	;?>$	Terminal
$<close_tag><stmt_list'>	?>$	R18 Ɛ
$<close_tag>	?>$	R7
$<close_tag'>?>	?>$	Terminal
$<close_tag'>	$	R9 Ɛ
$	$	Success

If the input string is matched with given grammar rules, the correct message is displayed as a parse tree as shown in Figure 2. If the input string is not matched with the grammar rules, the error messages and location errors are displayed.

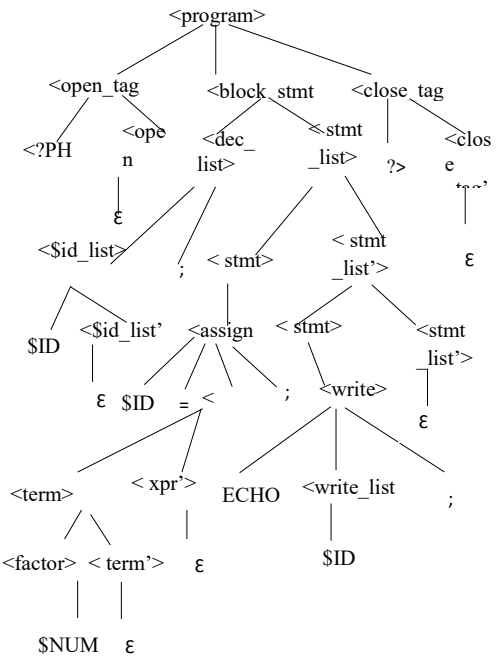

Figure 2. Parse tree for input sentence

5. Experimental Results

We show some input simple PHP program that is used for performance analysis. For evaluating purpose, different number of simple programs collecting from PHP programming tutorial site and PHP programming book are used as test set. The parser is tested on 30 programs. The performance of the system was good in all experiment scenarios for the various simple PHP programs. After parsing the system using the proposed grammar rules, it has been seen that the system can easily generate the parse tree for an input program if the program syntax structure satisfies the given grammar rules. Otherwise, it gives output as an error. Table 7 shows the performance of the parsing successful rate result

Table 7. Success rate for simple php program

Type	Total number of program(T)	Correct (N)	success rate A=(N/T)*100 %
Simple php program	30	24	80

Our result can compare with four related works. J.A.A1-Taani et al. [1] reported that 94.3% of 70

sentences were parsed successfully using efficient method, top down chart parser. W. W. Thant et al. [2] reported about 90.6% parsing accuracy on Myanmar sentences using function tags. K. M. A. Hasan et al. [3] reported an average accuracy of 78.2% for recognizing Bangla grammar using Predictive Parser. We reported that 80% of 30 programs were parsed successfully using efficient top down parsing approach. W. W. Thant [5] used 2200 Myanmar sentences using the Naïve Bayes theory, which gave an average accuracy of about 89.4%.

6. Conclusion

We proposed a system that can effectively investigate the syntax errors of the PHP programming language by using the top-down parsing approach. In the future, we will extend our system to be useful in many programming languages.

Acknowledgments

I would like to thanks all people who directly and indirectly contributed towards the success of this paper for the support, encouragement, useful suggestions, valuable guidance and help in preparing this paper.

References

[1] JAhmad A1-Taani, Mohammed Msallam and Sana Wedian, "A Top-Down Chart Parser for Analyzing Arabic Sentences", *The international Arab Journal of Information Technology*, Vol.9,No.2, March 2012, pp. 109-116.

[2] Win Win Thant, Tin Myat Htwe and Ni Lar Thein, "Context-free grammars Based Top-Down Parsing of Myanmar Sentences", *International Conference on Computer Science and Information Technology (ICCSIT'2011)*, Pattaya, Dec 2011, pp. 71-75.

[3] K.M.Azharul Hasan, AI-Mahmud, Amit Mondaal and Amit Saha "Recognizing Bangla Grammar Using Predictive Parser", I*nternational Journal of Computer Science &Information Technology (IJCSIT)*, Vol.3, No.6, Bangladesh, Dec 2011, pp. 61-73.

[4] Rachana Rangra and Madhusudan, "Basic Parsing Techniques In Natural Language Processing", *International Journal of Advances in Computer Science and Technology*, Vol.4,No.3, India, March 2015, pp. 18-22.

[5] Win Win Thant, Tin Myat Htwe and Ni Lar Thein, "Parsing of Myanmar Sentences with Function Tagging", Yangon, Myanmar.

[6] Bala sundara Raman L, Ishwar S and Sanjeeth Kumar Ravindranath, "Context-free grammars for Natural Language Constructs An Implementation for Venpa class of Tamil Poetry", India, 2003,pp.128-136.

[7] https://www.tutorialspoint.com/

[8] Robert W. Sebesta, *Concepts of Programming Languages*, Tenth Edition.